Software Architecture for Large-Scale, Distributed, Data-Intensive Systems

Chris A. Mattmann[*,†], Daniel J. Crichton[*], J. Steven Hughes[*], Sean C. Kelly[*] and Paul M. Ramirez[*,†]

[*]*Jet Propulsion Laboratory*
California Institute of Technology
Pasadena, CA 91109, USA
{mattmann,dcrichton,jshughes,sean.kelly,pramirez}@jpl.nasa.gov

[†]*Computer Science Department*
University of Southern California
Los Angeles, CA 90089, USA
{mattmann,pmramire}@usc.edu

Abstract

The sheer amount of data produced by modern science research has created a need for the construction and understanding of "data-intensive systems", large-scale, distributed systems which integrate information. The formal nature of constructing such software systems; however, is relatively unstudied, and has been a large focus of the super-computing and distributed computing communities, rather than the software engineering communities. These data-intensive systems exhibit characteristics which appear fruitful for research from a software engineering, and software architectural focus. From our experience, the methodologies and notations for design and implementation of data-intensive systems look to be a good starting point for this important research area. This paper presents our experience with OODT, a software architectural style, and middleware-based implementation for data-intensive systems developed and maintained at the Jet Propulsion Laboratory. To date, OODT has been successfully evaluated in several different science domains including Planetary Science with NASA's Planetary Data System (PDS) and Cancer Research with the National Cancer Institute (NCI).

1. Introduction

Science data management has seen an enormous increase in recent years, both in terms of the *quantity* of data generated, and in the *complexity* of the data itself. With the advent of new technologies and paradigms such as Grid-based systems [1,20], distributed-object middlewares [2,21] and wrapper-based information extraction tools [3], there has been a change in practice from developing one-off data management solutions which manage data independently (with little attention paid to future interoperability with other systems), to developing architectures and middlewares which are able to integrate and *re-use* existing data *resources*, such as web sites and legacy databases. This potential for re-use allows one to both imagine and realize the construction of large-scale distributed data management systems, whose main purpose is to *query, locate, access, process* and *distribute* data for potential users.

The *data-intensive* nature of these systems; however, shifts the focus from traditional software engineering methodologies and practices to believing that hardware, processing power, parallelism, and technology alone will be enough to construct software which can:

1. Access data in legacy data resources
2. Discover data which may be unknown at system run-time
3. Correlate the different data models which describe each data resource
4. Integrate the software interfaces to the legacy data resources.

Existing work in data-intensive systems severely lacks any common software engineering design constructs and relations between architectural components and implementation-level decisions. Our work has focused on this very area, the design and construction of large-scale data-intensive systems. We have developed an architectural style for data-intensive systems which allows software developers to construct architectural designs which address issues 1 through 4 above. In addition, a middleware implementation framework of the architectural style has been constructed and deployed. We have coined our architecture, and subsequent middleware implementation, OODT (*Object Oriented Data Technology*).

Section 2 describes the OODT architectural style. Section 3 covers the OODT middleware; a java based reference implementation of the OODT architectural style, and ties the middleware back to the architectural style assumptions and properties. Section 4 presents two deployments of the OODT style and middleware in the planetary science and cancer research domains, which both share many of the same data-intensive system requirements presented above. Section 5 surveys related work in data-intensive systems. Section 6 addresses open issues within OODT. Section 7 rounds out the paper.

2. An Architectural Style for Data-Intensive Systems

One method of defining an architectural style has been to define the primitive building blocks, software components and connectors, and then define legal topologies of these primitive building blocks needed to construct systems that implement the style itself [4, 16, 19]. Using these core definitions, system implementations are constructed in the style, and analyzed to verify that key assumptions of the style at *design-time* are validated by the implemented system at *implementation time*. We use this method to construct the basis for the OODT architectural style. To begin, we define the terms used across the definitions of the style as follows:

- *Data Source* – Any entity that can provide data (e.g. a database, a web site, a data producing software system and so on)
- *Metadata* – Describes the content, quality, conditions, and other characteristics of data ("data about data").
- *Data Model* – A description of the data provided by a data source, i.e. the metadata for all data in a data source.
- *Data element* – Metadata that describes one facet of the data from a particular data source. For example, in the case of book data, *Title*, would be one possible data element.
- *Resource* – A data source, a pointer to a unit of data, or a unit of data which can be queried, followed or retrieved respectively.
- *Data Product* – A unit of data or metadata which can be retrieved from a data source.

2.1 Components

The core components for OODT are *Product Servers*, *Profile Servers* and *Query Servers* [5]:

- **Product Servers** abstract away data source interfaces (such as SQL, File Systems, HTTP) into source-independent interfaces for retrieval of data which satisfy the user queries.
- **Profile Servers** serve back metadata [5] in the form of resource profile data structures, which provide data for deciding what resources satisfy particular queries.
- **Query Servers** accept user queries, and then use profile servers to determine what resources to query and collect in order to satisfy the query and return data products. The Query Server then aggregates and returns back the federated data products to the caller.

2.1.1 Product Servers

Product Servers abstract away data source dependent interfaces to data by wrapping [6] the data source interface such that it can support OODT queries. OODT queries are in the format of one or more un-ordered (*keyword comparison operator value*) predicates which themselves are joined by zero or more logical operators (i.e. AND, OR, etc). Comparison operators are operators such as "EQUALS, GREATER THAN and LIKE".

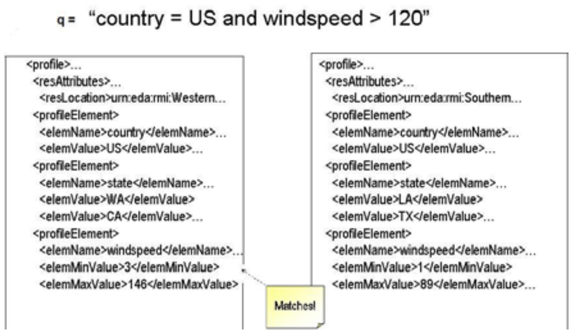
OODT queries are posed against a set of *Common Data Elements* (which formulate a high-level data model for the data-intensive system, referred to from here on as a *Data Dictionary*), and thus the set of allowable keywords in each predicate comes from the set of the common data elements in the system's data dictionary. Each product server contains zero or more *Query Handlers* which serve back different types of data products from each data resource. Our rationale behind having more than one query handler per product server is due to the fact that there may be multiple types of data products that are stored in the same data source. A case like this could be imagined in a relational database, which stores both binary images of Mars, and also metadata containing information about the space instrument which captured each image. Each query handler contains an translation function *t(q)*, which maps the query from the data dictionary domain, to a query in the domain which the data source understands (i.e. maps common data elements against *Source Data Elements*, data elements in the data source domain). This function is represented by:

$$(1) \quad t(q) = q \in D_1, q \longrightarrow q', q' \in D_2$$

The main goal is to take the abstract query against the common data elements, and map that into something that the underlying data sources that are integrated understand. An example of this translation would be as follows. Given an OODT query such as "BookType = Fiction AND Author = Cricthon" and a relational database containing book data, the OODT query is translated into: `select books.* where book.type = 'fiction' and book.author = 'crichton'`.

2.1.2 Profile Servers

Profile Servers respond to OODT queries, and return descriptions of resources which can satisfy the system query. To achieve this functionality, profile servers store resource *Profiles* which describe the data elements and semantic relationships that each data resource supports. In this fashion, a profile server can receive an OODT query, and match each keyword data element and associated semantic constraints to a particular resource that a profile describes, and return the matching profile(s) which satisfy the query. This scenario is illustrated in Figure 1.



```
q = "country = US and windspeed > 120"
```

```
<profile>...
  <resAttributes>...
    <resLocation>urn:eda:rmi:Western...
  <profileElement>
    <elemName>country</elemName>...
    <elemValue>US</elemValue>...
  <profileElement>
    <elemName>state</elemName>...
    <elemValue>WA</elemValue>
    <elemValue>CA</elemValue>...
  <profileElement>
    <elemName>windspeed</elemName>...
    <elemMinValue>3</elemMinValue>
    <elemMaxValue>146</elemMaxValue>
```

```
<profile>...
  <resAttributes>...
    <resLocation>urn:eda:rmi:Southern...
  <profileElement>
    <elemName>country</elemName>...
    <elemValue>US</elemValue>...
  <profileElement>
    <elemName>state</elemName>...
    <elemValue>LA</elemValue>
    <elemValue>TX</elemValue>...
  <profileElement>
    <elemName>windspeed</elemName>...
    <elemMinValue>1</elemMinValue>
    <elemMaxValue>89</elemMaxValue>...
```

Figure 1. Query q and matching Profile of Resource

The profile structure is based on two internationally accepted standards, ISO/IEC 11179 [7] to describe the structure of data elements, and the Dublin Core [8] set of data elements, which are used to describe *any electronic resource*. Our choices for these standards are explained further in [5] and beyond the scope of this paper, but we refer to them here in order to motivate the discussion of the profile structure. The profile structure contains metadata about the profile itself, such as a unique ID, a name for the profile, and an author of the profile. We refer to this profile metadata as *Profile Attributes*. The next piece of the metadata that the profile contains is an implementation and extension of the Dublin Core elements to describe the resource that the profile describes. These elements include *Creator*, *Name* and *Author*. Further, our Dublin Core extensions include the addition of specific data elements to describe resources, (1) a *Resource Location* element to describe a resource's location (physical locations such as URLs are supported, as well as OODT-specific URNs [10] which we detail later) and (2) A taxonomy to categorize the type of resource (we refer to this element as *Resource Class*). Resource Class can be a product server, another profile server, or the data product itself. We refer to this extended Dublin Core metadata as *Resource Attributes*. The final piece of metadata that we include in profiles is domain-specific data elements which describe the resource in its originating domain. For example, in the Planetary Data System [11], the data element "Target Name" is used to describe each data product's originating "target" (i.e. planet). These domain-specific data elements are called *Profile Elements*. The profile metadata is summarized in Table 1 below.

Table 1: Metadata Stored in each Resource Profile

Metadata Type	Description of Metadata	Example
Profile Attributes	Metadata describing the profile itself	ProfileId
Resource Attributes	Dublin Core based metadata describing the resource that the profile describes, extended to include elements such as *Resource Location* and *Resource Class*	Creator, Resource, Class, Title
Profile Elements	Domain-specific Metadata describing the resource which the profile describes	Target Name

Similar to product servers, each profile server contains zero or more query handlers which accept an OODT query, and then query for the profiles which the profile server manages. The profile query handlers are used to abstract away the underlying data source which stores the resource profiles and return the profiles much the same way that product servers return data products.

2.1.3 Query Servers

Query Servers are responsible for "putting it all together". Initially a query server is bootstrapped with an initial set of root profile servers, much like Berkeley's DNS [9]. The query server contains a list of collected data products which is populated using the following process. Upon receiving an OODT-style query, the query server proceeds to query each one of the profile servers it knows of to retrieve the profiles which match resources which satisfy the query. This querying of profile servers is performed by *Querier* components. Each querier component is given a location for a profile server, and is able to query the profile server, and retrieve the list of profiles (if any) that describe resources that could satisfy the particular query. Each profile returned contains a resource class attribute (recall section 2.1.2) which is examined to determine where to go next in order to retrieve the requested data products. There are three cases which occur during this examination:

- **The Resource Class points to another Profile Server:** In this case, the querier creates another instance of a querier component, seeded with the returned profile server resource location attribute as its root profile server.
- **The Resource Class points to a Product Server:** In this case, the querier creates a *Product Client* component instance, which it seeds with the resource location attribute of the returned profile. The product client component retrieves the data products from the product server, and then the product client returns the data products back to the querier component. The querier component then adds its data products to the collected

data products list in the query server, and ends its thread lifespan.

- **The Resource Class points to an individual data product:** In this case, the querier retrieves the individual data product and adds it to the collect data products list in the query server and ends its component lifespan.

Once the Querier components have ended their respective lifespan, the Query Server is done collecting data products in its collected list. The collected data product list is then returned to the original user query.

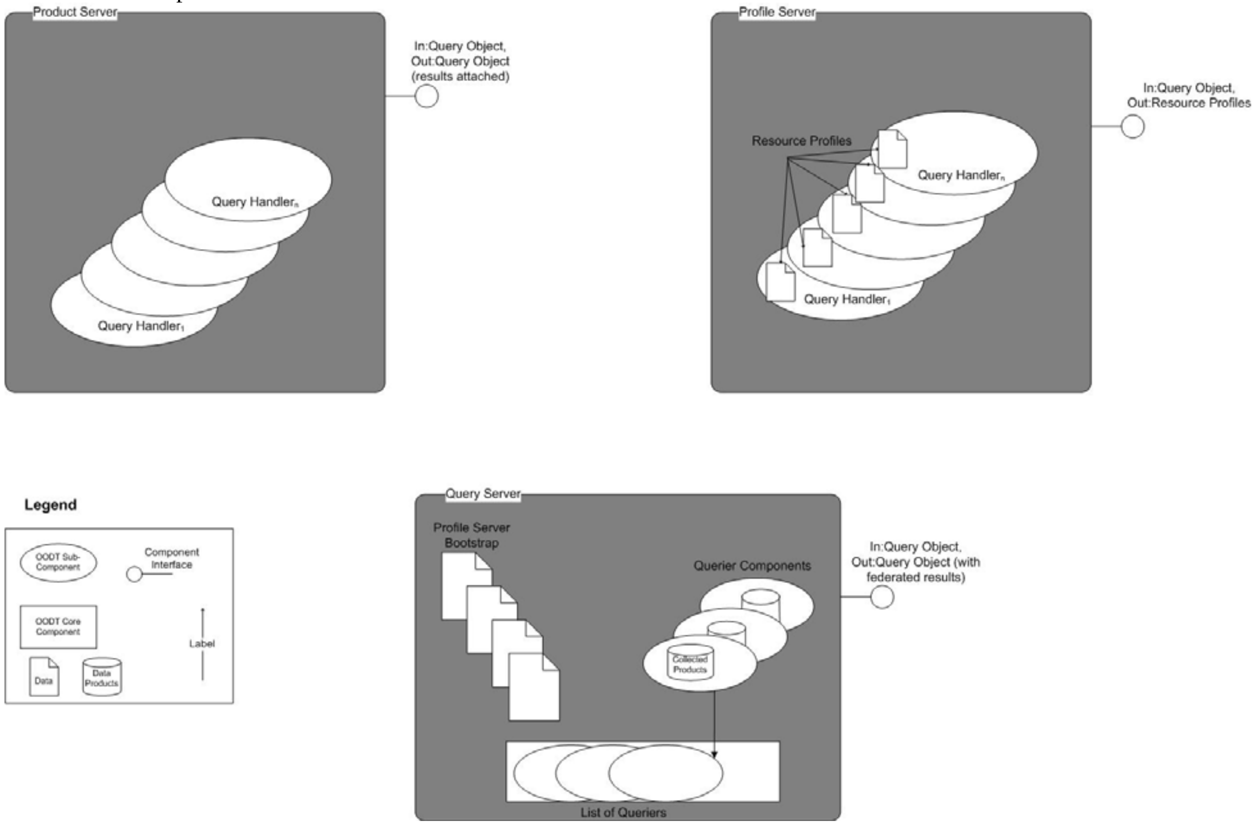


Figure 2. The OODT Software Components

2.2 Connectors

OODT supports one type of connector, a *Messaging Layer*, which allows OODT components to communicate with each other. The main form of communication between OODT components is the *Query Object*. A query object contains the query q (recall Section 2.1.1), and the query result, i.e. the collected list of data products (recall Section 2.1.3). The query object also contains a set of associated metadata regarding the query, such as *Maximum Accepted Results*, *Accepted Mime Types* (of data products within the collected data products list), and *Query Description*.

2.2.1 Messaging Layer

The OODT Messaging Layer's responsibility is to route query objects between query server, product server, and profile server components, and return the query object to the correct *User* (typically the user which contacted the query server). The messaging layer essentially provides a message bus which allows many Query Objects to be transferred between OODT components in a variety of different messaging Protocols including *Unicast*, (query object sent from an originating OODT component to one other component), *Multicast* (query object sent from an originating OODT component to a group of other OODT components defined by some common attribute, e.g. a URN [10].) and *Broadcast* (query object sent from an originating OODT component to all other OODT components known to the originator).

2.3 Configurations

OODT Configurations are constructed from the core software components described above. First each data source that OODT will integrate is identified and a product server component is instantiated for each data source to be integrated. Once all the product servers are attached to the data sources, profile server components are created to serve back profiles which point to the product servers created in the previous step. Last, one or more query server components are created, and seeded with the list of profile servers created in the previous step, and the query servers can then begin accepting queries and returning data.

Section 4 presents two deployments (and subsequent *architectural configurations* of OODT components) of the OODT architectural style in two diverse domains, Planetary Science, and Cancer Research.

3. Architecture-based Middleware

The core OODT middleware has been implemented in Java, using Sun's Java Development Kit. Programmers extend OODT core component classes such as *Profile Servers*, *Product Servers*, and *Query Servers*.

For the implementation of the product server component, programmers are required to write *QueryHandler* classes which implement the `jpl.oodt.product.[layer].QueryHandler` interface. *Layer* represents the type of messaging layer connector the programmer wants to use. Currently, CORBA and RMI exist, and we are working on a SOAP-based messaging layer at the time of writing this paper. The interface method to a query handler for a product server is the *query* method. The query method takes as input a query object (recall Section 2.2) which can be represented and serialized into XML. The query object contains the query *q* (recall Section 2.1.1), and the associated query *metadata* (recall Section 2.2). After retrieving the requested data products, the product server query method returns the query object back to the user, this time including any results which satisfied the query to the underlying data resource. The query interface method thus becomes the abstract translation function *t(q)* discussed in section 2.1.1. We present below an XML DTD definition of our query object structure.

```
<!ELEMENT query
      (queryAttributes,queryResultModeId,
        queryPropogationType,queryPropogationLevels,
        queryMimeAccept*,queryMaxResults,
        queryResults,queryKWQString,
      queryStatistics?, querySelectSet,
        queryFromSet,queryWhereSet,
      queryResultSet)>
    <!ELEMENT queryAttributes
      (queryId, queryTitle*, queryDesc*, queryType*,
       queryStatusId*, querySecurityType*,
       queryParentId*, queryChildId*,
       queryRevisionNote*,queryDataDictId*)>
    <!ELEMENT queryStatistics (statistic*)>
    <!ELEMENT querySelectSet
       (queryElement*)>
    <!ELEMENT queryFromSet
       (queryElement*)>
    <!ELEMENT queryWhereSet
       (queryElement*)>
      <!ELEMENT queryElement
        (tokenRole*, tokenValue*)>
    <!ELEMENT statistic (url, time)>
    <!ELEMENT queryResultSet
       (resultElement*)>
      <!ELEMENT resultElement
        (resultId*, resultMimeType*,
         profId*, identifier*, resultHeader,
         resultValue*)>
        <!ELEMENT resultHeader (headerElement*)>
        <!ELEMENT    headerElement    (elemName,    elemType?,
elemUnit?)>
```

The product server itself supports the exact same query method as its query handler component(s). When a query comes into the product server (via CORBA or RMI), the product server creates worker threads which each are given a particular query handler object reference, as well as a reference to the xml-based query object (containing the overall result list). In parallel, each worker thread invokes its query handler's query method, and passes it the reference to the product server's xml query object, so that each query handler upon return will aggregate the list of data products into the results section of the query object.

The profile server component implementation centers around the use of XML-based profiles, which profile data resources (recall Section 2.1.2 and Table 1). An XML DTD for the profile structure is given below:

```
<!ELEMENT profiles
(profile*)>
<!ELEMENT profile
(profAttributes,
resAttributes,
profElement*)>
<!ELEMENT profAttributes
(profId, profVersion?, profType,
profStatusId, profSecurityType?, profParentId?,
profChildId*,
profRegAuthority?, profRevisionNote*, profDataDictId?)>
<!ELEMENT resAttributes
(Identifier, Title?, Format*, Description?, Creator*,
Subject*,
Publisher*, Contributor*, Date*, Type*, Source*,
Language*, Relation*, Coverage*, Rights*,
resContext+, resAggregation?, resClass, resLocation*)>
<!ELEMENT profElement
(elemId?, elemName, elemDesc?, elemType?, elemUnit?,
elemEnumFlag, (elemValue* | (elemMinValue, elemMaxValue)),
elemSynonym*,
elemObligation?, elemMaxOccurrence?, elemComment?)>
```

The profile server component is implemented by writing profile server query handlers, which return back resource profiles in the above XML format. The profile server itself provides two interface methods, (1) A query method which accepts an xml based query object and returns resources profiles which can satisfy queries against the data dictionary data elements used in query object's query string and (2) A getProfile method which accepts a profile Id, and returns the exact profile which matches the given id. The profile server query method returns profiles by querying (in parallel using the worker thread method discussed for the product server implementation) its respective profile query handlers. Each profile query handler implements the same 2 methods as its parent profile server, which allows each query handler to return profiles asynchronously back to the parent.

Upon creating query handlers that return both data products, and profiles, the method for linking the Query Handlers to the product and profile servers respectively can be performed via XML configuration files, or at system-runtime. The XML configuration file essentially sets each product and profile server specified to have the associated query handlers, and instantiates each respective component. After instantiation, the system determines what type of messaging connector is used (either RMI or CORBA), and then registers the profile and product server components with the appropriate registry (in RMI, Sun's RMI Registry can be used, in CORBA, the CORBA Name Service is used). Query server components can then be instantiated and given the list of profile servers which exist in the system. Each profile, product, and query Server component is identified in the form of URNs [10], which uniquely identify the distributed component in the respective registry type. Our URN's are in the form of `urn:oodt:[layer]:[component-name]`. The respective layer registry enforces the requirement that the combination of [layer]: [component-name] must be unique when components register at run-time.

The implementation of the query server component typically comes last, which is directly tied to its architectural style notion of "bringing it all together". The query server component is seeded (either at run-time via configuration files or compile time via code) with a set of root profile server components (identified by URNs described in the above paragraph). The query server component supports one interface method, a query method. The query method accepts a query object as a parameter, and returns back the same query object this time containing the federated results that it was able to collect from the OODT connected system. To perform this result collection, the query component instantiates a base set of querier threads (recall Section 2.1.3) which are each given a root profile server in the query server component's initial list. Each querier thread proceeds in parallel to query its respective profile server and retrieve resource profiles by which to discover locations of resources which satisfy the original query. The querier threads use the algorithm described in Section 2.1.3 to examine each resource profile's resource class attribute to determine how to go about reaching an eventual data product. Once all the original (and any additional Querier threads which were created) are finished collecting products, the method returns the query object with all the collected results.

4. Experience and Evaluation

In this section, we discuss a deployment of the OODT middleware in the *Planetary Science* domain, with NASA's Planetary Data System [11]. We follow by presenting another OODT deployment supporting Cancer Research with NCI's Early Detection Research Network (EDRN) [12].

4.1 Planetary Data System

The Planetary Data System (PDS) manages and archives planetary science data for NASA's Office of Space Science. It has been in existence since the late 1980's and to this point has collected approximately six terabytes of data, which pre-OODT was distributed and archived for scientists on CD and DVD media. The PDS is divided up into eight discipline "nodes", which are geographically distributed across the country (shown in Figure 3). Each node is responsible for managing and distributing its own particular planetary science data which is cumbersome due to their geographically diverse locations, and different methodologies (discussed below) for storing and accessing their data.

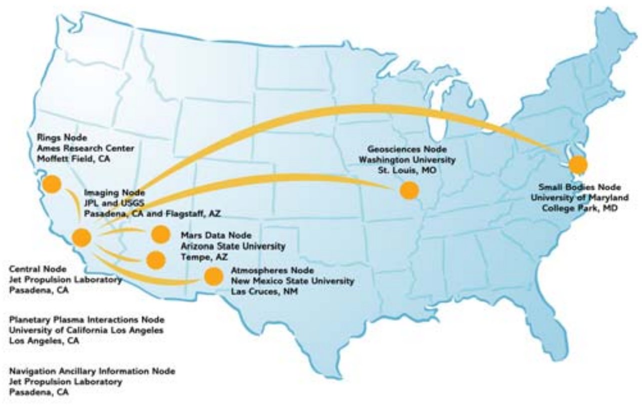


Figure 3. PDS Geographic Diversity

The OODT middleware was used in 2002 to deploy an infrastructure for the distribution of PDS data from all eight nodes via the internet. OODT product server components wrapped data resources at each PDS node (a total of 8 active Product Servers) and a master set of 5 Profile Servers were used to profile the data resources exposed by the OODT product Server components. To tie everything together, a web search page was constructed to allow a user to pose a federated query across the entire PDS, and receive back the federated PDS data. The web portal search page poses its queries to an OODT query server, seeded with the master set of profile servers mentioned above.

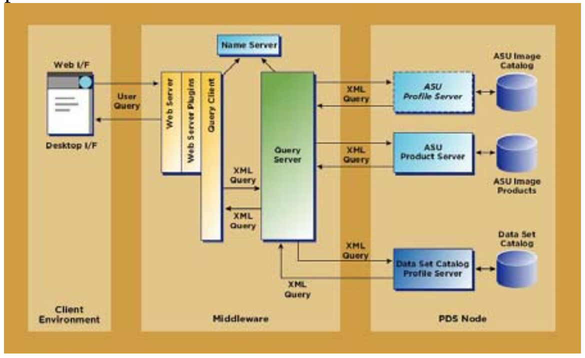
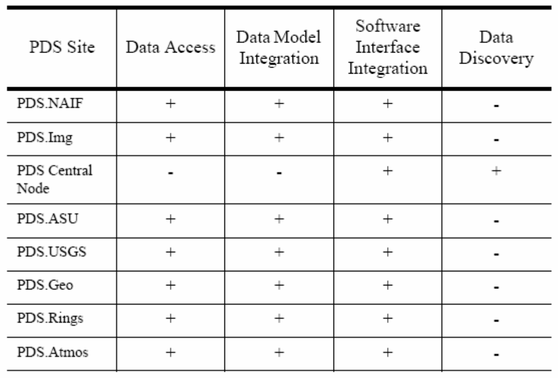


Figure 4. PDS deployment using the OODT style and middleware

To begin, we evaluate the PDS deployment against the 4 initial data-intensive issues (recall Section 1) and present our evaluation in Table 2 below. + indicates full support for the data-intensive issue, - indicates anything less than full support.

Table 2: PDS Evaluation against data-intensive system issues.



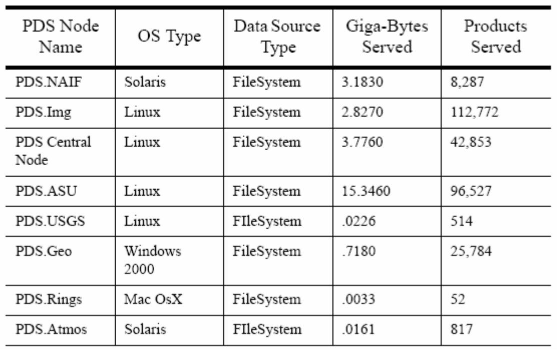
PDS Site	Data Access	Data Model Integration	Software Interface Integration	Data Discovery
PDS.NAIF	+	+	+	-
PDS.Img	+	+	+	-
PDS Central Node	-	-	+	+
PDS.ASU	+	+	+	-
PDS.USGS	+	+	+	-
PDS.Geo	+	+	+	-
PDS.Rings	+	+	+	-
PDS.Atmos	+	+	+	-

Further, we evaluate the PDS deployment against the following additional dimensions in order to highlight both the heterogeneity of the distributed system, along with its data-intensive nature:

1. Operating System that each PDS node is running
2. Underlying data source that each product server integrates
3. Amount of data served back by each PDS node (including both resource profile queries, and product queries)
4. Number of data products transferred at each PDS product server

Our results are summarized in Table 3.

Table 3: PDS Heterogeneity and Data-Intensive Nature

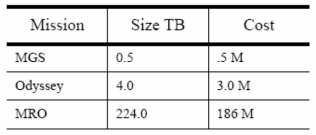


PDS Node Name	OS Type	Data Source Type	Giga-Bytes Served	Products Served
PDS.NAIF	Solaris	FileSystem	3.1830	8,287
PDS.Img	Linux	FileSystem	2.8270	112,772
PDS Central Node	Linux	FileSystem	3.7760	42,853
PDS.ASU	Linux	FileSystem	15.3460	96,527
PDS.USGS	Linux	FIleSystem	.0226	514
PDS.Geo	Windows 2000	FileSystem	.7180	25,784
PDS.Rings	Mac OsX	FileSystem	.0033	52
PDS.Atmos	Solaris	FIleSystem	.0161	817

Since 11/15/2002, the PDS Product Server components have transferred 25GB of PDS Data Products to Scientists across the country (note that the 25GB so far are not including the full distributions of Mars Global Surveyor, Mars Odyssey, and upcoming MRO missions because all of the data has not been delivered to the PDS yet)

Table 4 helps to illustrate the benefit of OODT in the PDS. Before OODT, PDS data was distributed via CD and DVD media, and the PDS nodes had to mail out CDs and DVDs to scientists which contained the planetary science data from the missions which they desired to study. In part, this is due to the heterogeneity of the data stored at each PDS node, and each node's neglect to focus on data system interoperability software architecture. Since OODT, the data can now be distributed, and accessed via the web, as if the data stored in a single virtual data source. PDS nodes can also download the OODT middleware implementation framework and create their own OODT components which will plug into the existing middleware infrastructure, which will help to discourage construction of "one-off" data systems which have to be integrated at a later time. In Table 4, we show 3 NASA/JPL missions, Mars Global Surveyor (MGS), Mars Odyssey (Odyssey), and Mars Reconnaissance Orbiter (MRO) and the amount of science data (in terabytes) that each mission was set to produce. The last column, Cost, shows the estimated cost for distributing those PDS products on DVD media, and mailing out the PDS data to the scientists, using the old data delivery method that NASA was forced to use because of the PDS heterogeneity. The upcoming MRO mission is set to produce a *un-precedented* 224 *terabytes* of data which pre-OODT would have cost NASA an estimated 186 million dollars to distribute to the planetary scientists who need the data to perform their research.

Table 4: Cost-comparison of upcoming Mission data using old delivery method.



Mission	Size TB	Cost
MGS	0.5	.5 M
Odyssey	4.0	3.0 M
MRO	224.0	186 M

4.2 Early Detection Research Network

The National Cancer Institute's Early Detection Research Network [12] is a network of over 30 cancer research sites participating in research geared towards the early detection of cancer. Particularly of interest to EDRN is *cancer biomarker data* [12]. Similar to the PDS, each EDRN site is geographically distributed across the United States, and each site contains data systems which do not interoperate and commingle with the other EDRN sites' data systems. The ability to correlate this information is critical to cancer research in that it has been shown that as study volumes increase, so does the rate of scientific discovery [12]. Also, in terms of validating and testing biomedical data, it is important to compare and contrast similar data at different EDRN sites.

The OODT middleware is currently supporting EDRN by providing the middleware infrastructure to integrate the distributed cancer research data located across EDRN's sites. Product servers at 9 of the current EDRN locations wrap site specific data sources, and expose the cancer data to the overall system. A Profile server, located at Fred Hutchinson Cancer Research Center in Seattle provides the means for discovery of the existing product servers in the network. A portal web page, the ERNE (EDRN Resource Network Exchange) [12] provides web-based access to the OODT-based query system.

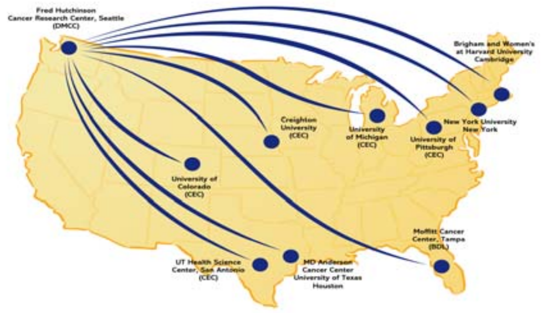


Figure 5. EDRN Geographic Diversity

Below, we present the EDRN deployment which was built using the OODT style software architecture. OODT currently is deployed at 9 of the existing EDRN sites, 8 of which are shown in the diagram below.

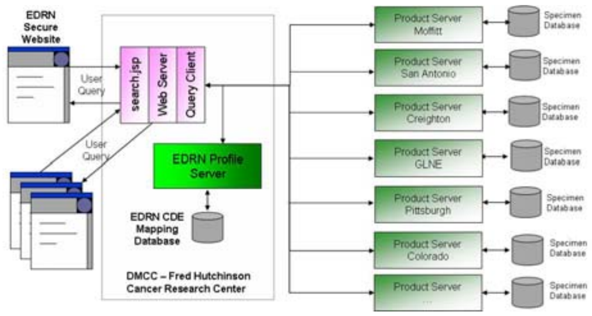


Figure 6. EDRN deployment using OODT style and middleware.

We begin by evaluating the EDRN deployment against the initial data-intensive issues (recall Section 1) and present our evaluation in Table 5 below.

Table 5: EDRN Evaluation against data-intensive system issues.

EDRN Site	Data Access	Data Model Integration	Software Interface Integration	Data Discovery
Moffit	+	+	+	-
San Anto-nio	+	+	+	-
Creighton	+	+	+	-
Pittsburg	+	+	+	-
DMCC	-	-	+	+
Colorado	+	+	+	-
GLNE	+	+	+	-

Each EDRN site besides the DMCC (Data Management Coordinating Center) at Fred Hutchinson Cancer Research Center in Seattle employs a product server for Data Access (the DMCC hosts the system's profile server) and therefore each EDRN site provides some level of Data Model Integration (via each product server's t(q) function). Since each product and profile server employs a standard software interface, each site provides a level of software interface integration. Finally, Data Discovery is facilitated by the DMCC site because it contains the profile server which contains the initial list of product servers which are available to satisfy resources.

To conclude our evaluation, we further evaluate our EDRN deployment against the dimensions used in Table 3 to evaluate system heterogeneity and data-intensive nature. The results of our evaluation are shown in Table 6.

Table 6: EDRN Heterogeneity and Data-Intensive Nature

EDRN Node Name	OS Type	Data Source Type	Giga-Bytes Served	Products Served
Moffit	Windows 2000 Server	MS SQL (ODBC)	.0096	477
San Anto-nio	Windows 2000	MS SQL (ODBC)	.0016	449
Creighton	Windows 98	MS SQL (ODBC)	.00004	459
Pittsburg	Windows 2000	MS SQL (ODBC)	N/A	494
Colorado	Windows 2000	MS SQL (ODBC)	.0011	511
GLNE	Windows 2000	MS SQL (ODBC)	.0028	459

Our evaluation is shown for 6 of the EDRN sites of which we had recorded data for. The EDRN data above has been recorded since 1/14/03. N/A indicates that data was not available for a particular site.

5. Related Work

The work-in-progress by Kolp and Mylopoulos [13] models information systems using business-organizational structure as an architectural style, and argues that Multi-agent Systems (MAS) should be considered as a construct for information system architectural styles. They primarily focus on the domain of business information systems. They are also restricting their study to requirements engineering and conceptual architecture for such systems. On the other hand, our goal is to provide end-to-end software development solutions, spanning the entire software engineering lifecycle for data-intensive systems.

Gomaa et al. [14,15] present a novel architecture for describing large-scale data-intensive information systems, specifically applied to NASA's EOSDIS science domain. This work is closely related to ours, but focuses only on a single style – federated client-server [16]. Furthermore, they present no fine-grained mapping of the conceptual architecture to the deployment architecture, or middleware-based solutions for implementing the data-intensive system. The EOSDIS system also suffered from many design flaws, which are discussed in [17].

Moore et al. [18] define Data-intensive systems as systems which are IO-bound. They specifically describe the SDSC[1] Storage Request Broker (SRB) and how it can be used to abstract domain-specific data sources using a layered architectural style [19]. Moore et al. also presents a table[2] defining a basic set of application requirements for data-handling environments. In contrast, our approach aims to perform such work throughout the software engineering lifecycle for data-intensive systems. Furthermore, as discussed below in the context of grids, layered service architectures provide no insight into the effective topologies of a system's constituent software components and connectors. Layered service architectures also typically provide little guidance about the legal behaviors of components, as laid out in deployment architectures.

Recent work in the Grid Community [1] has characterized a class of distributed data interoperable systems as Data Grids [20]. Data Grids are discussed architecturally in terms of a layered services architecture [19], and web-serviced based [21] middleware implementation. This approach is similar to ours in the sense that the architecture is developed first, and then middleware is instantiated to implement the architectural constructs. However, there has been no focus on mapping system requirements to architectural components and, furthermore, to implementation-level artifacts in the data grid community. In fact, it has been recognized that initial efforts on data grids (and the grid community as a whole) have focused on "getting it to work" rather than system scalability, evolution, or design [22].

Singh et al. [23] define a metadata catalog service (MCS) component, but do not describe its relationship in full to the layered services architecture adopted by the grid community (e.g., the Globus toolkit [1]) nor to its deployment and interactions with other grid components. Singh et al. focus on the MCS component's scalability, which is indicative of the grid community's tendency to focus on super-computing challenges, as opposed to effective software engineering methodologies. Our work is geared towards uniting formal software engineering principles with the design of data-intensive information integration systems such as data grids.

Sun's Enterprise Java Beans (EJB) [24] are extensible software components which are tightly coupled to an underlying middleware infrastructure providing network services, secure transactions and component discovery. EJB are popular in industry as implementation-level constructs supporting the development of distributed systems. EJB provide no native support; however, for data-intensive issues such as *data access*, *data discovery* and *data model correlation* that our work supports. Indeed, our work has focused on using middleware implementation infrastructures such as EJB to support basic component services, and then providing middleware-specific implementations of Product Servers, Profile Servers, and Query Servers to support data-intensive functionality. EJB could be envisaged as an implementation-specific OODT *messaging layer* connector.

6. Open Issues

There remain several open issues with OODT that we will address briefly in this section. First and foremost, architecting, and deploying OODT software architectures, and subsequent middleware is very *programmer-intensive*. By "programmer-intensive" we mean that a programmer is required to be "in the loop" in order to successfully deploy and architect these systems. This is due to the fact that the programmer is responsible for translating OODT architectural constructs into extensions of our existing middleware framework (recall Section 3). Typically, a programmer will be involved in the early phases of the software process, helping to gather requirements, and translate requirements into some tailoring, configuration and deployment of existing OODT code.

One way of addressing problems like these has typically been to provide architectural design tools, such as UC Irvine's xADL [25] for software designers to create architectural diagrams, and then have a way of mapping those architectural diagrams to software implementation and code deployment. There is existing work in this area [26], and we aim to research and construct tool support to model and deploy architectures and software systems in the OODT-style.

The OODT middleware also assumes a reliable network is present in order for the Product, Profile, and Query Servers to communicate across. We currently have no support for issues such as disconnected operation [27], and off-loading of data to support unreliable hosts. This type of fault-tolerance is crucial in data-intensive systems which may be deployed in unreliable environments.

7. Conclusions

We have presented OODT, an architectural style and middleware implementation for data-intensive systems. We feel that data-intensive systems have been a neglected area of research in the software engineering and software architecture communities, and we desire to apply formal software architectural methodologies to the design, implementation, and evolution of data-intensive systems. The OODT style and middleware was

[1] San Diego Supercomputing Center (http://www.sdsc.edu)
[2] Table 5.2 in [18]

developed at NASA's Jet Propulsion Laboratory and has been supporting NASA's Planetary Data System in the Planetary Science Domain. The OODT middleware also supports Cancer Research, and is currently deployed at the National Institute of Health's National Cancer Institute, supporting the EDRN (Early Detection Resource Network) task.

8. Acknowledgments

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

9. References

[1] The Globus Alliance. http://www.globus.org. January, 2004

[2] E. M. Dashofy, N. Medvidovic, and R. N. Taylor. Using Off-The-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA.

[3] C. A. Knoblock et al. Accurately and reliably extracting data from the web: A machine learning approach, *IEEE Data Engineering Bulletin*, 23 (4):33-41, December 2000.

[4] R. N. Taylor, N. Medvidovic et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pages 390-406 (June 1996).

[5] D Crichton, J.S. Hughes and S. Kelly. A Science Data System Architecture for Information Retrieval. In *Clustering and Information Retrieval*. Kluwer Academic Publishers. December 2003.

[6] H.M. Sneed. The Rationale for Software Wrapping. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, October 01-03, 1997, p. 303.

[7] ISO/IEC, Framework for the Specification and Standardization of Data Elements 11179-1, Specification and Standardization of Data Elements 11179, International Organization for Standardization, Geneva, 1999.

[8] DCMI, Dublin Core Metadata Element Set, Version 1.1: Reference Description, Dublin Core Metadata Initiative, 1999.

[9] Berkeley Internet Name Domain (BIND). http://www.isc.org/bind.html, 2004.

[10] RFC 3613. Definition of a Uniform Resource Name (URN) namespace, 2003.

[11] J. S. Hughes and S. K. McMahon. The Planetary Data System. A Case Study in the Development and Management of Meta-Data for a Scientific Digital Library. In *Proceedings of the European Conference on Digital Libraries (ECDL)*, 1998. pp. 335-350

[12] H. Kincaid, D. Crichton et al. A National Virtual Specimen Database for Early Cancer Detection. *In Proceedings of the 16th IEEE Symposium on Computer Based Medical Systems (CBMS)*. New York, New York, June 2003, p. 117.

[13] M. Kolp and J. Mylopoulos. Architectural Styles for Information Systems: An Organizational Perspective, Tropos Working Paper. University of Toronto, Department of Computer Science, January 2001.

[14] H. Gomaa, D. Menasce, and L. Kerschberg, A Software Architectural Design Method for Large-Scale Distributed Information Systems, *Journal of Distributed Systems Engineering*, 1996.

[15] L. Kerschberg et al. Data and Information Architectures for Large-scale Data Intensive Information Systems. In *Proceedings of the 8th International Conference on Statistical and Scientific Database Management*, Stockholm, Sweden, June 18-20, 1996

[16] R. T. Fielding. *Architectural Styles and the design of Network-based Software Architectures*, Ph.D. Dissertation, University of California, Irvine, 2000.

[17] A. T. Leath. NASA's Earth Science Programs come under Scrutiny. *The American Institute of Physics Bulletin of Science Policy News* Number 126: September 11, 1998. http://www.aip.org/enews/fyi/1998/fyi98.126.htm

[18] R. W. Moore et al. Data-Intensive Computing. In *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufman Publishers, 1999.

[19] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.

[20] A. Chervenak, I. Foster et al. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications*, 2000.

[21] Webservices.org. http://www.webservices.org, January, 2004.

[22] H. Casanova, Distributed Computing Research Issues in Grid Computing, *ACM SIGAct News*, Vol. 33, No. 3, 2002, pp. 50-70.

[23] G. Singh, S. Bharathi et al. A Metadata Catalog Service for Data Intensive Applications, In *Proceedings of the IEEE Conference on Supercomputing*, 2003.

[24] N. Medvidovic and N. R. Mehta. JavaBeans and Software Architecture. In *The Internet Encyclopedia*, Hossein Bidgoli (ed.), John Wiley & Sons, December 2003.

[25] E.M. Dashofy, A. Van der Hoek and R.N. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02), p.266*-267, 2002.

[26] N. Medvidovic et al. Software Architectural Support for Handheld Computing. Cover feature in *IEEE Computer*, September 2003.

[27] M. Mikic-Rakic and N. Medvidovic. Toward a Framework for Classifying Disconnected Operation Techniques. *In Proceedings of the 2nd International Workshop on Software Architectures for Dependable Systems (WADS'03)*, Portland, Oregon, May 2003.